4. Configure, Build, and Install the Toolkit

Introduction

This chapter describes how to configure and build the NCBI C++ Toolkit on various platforms. See also the Getting Started chapter for general overview and some details.

Chapter Outline

UNIX

- Supported UNIX platforms
- Building instructions

MS Windows

- MS Visual C++.NET 7.1
- MS Visual C++.NET 8.0
- Cygwin/GCC

Mac OS X

- Xcode
- GCC
- Code Warrior

UNIX

Supported UNIX platforms

Building instructions

- Configuration and Installation Script configure
- Structure of the Build Tree Produced by configure
- Running the configure Script
    - Getting Synopsis of Available Configuration Options
    - Debug vs. Release Configuration
    - Building Shared Libraries (DLLs)
    - Hard-Coding Run-Time DLL Path into Executables and DLLs
    - Multi-Thread Safe Compilation and Linking with MT Libraries
    - Building in the 64-bit mode
    - Automatic Generation of Dependencies (for GNU make Only)
    - Naming the Build Tree
    - After-Configure User Callback Script

- Tools and Flags
- Localization for the System and Third-Party Packages
- Prohibiting the Use of Some of the System and Third-party Packages
- Optional Projects
- Finer-grained Control of Projects: --with-projects
- Miscellaneous: --without-exe, --without-execopy, --with-lib-rebuilds(=ask)
- Compiler-specific configure Wrappers

- Quick Reconfiguration

Configuration and Installation Script *configure*

Different compilers compile C++ (and even C!) code differently; they may vary in the OS standard libraries and header files, completeness of the C++ implementation, and in compiler bugs. There are also different *make* and other tools and file naming conventions on different platforms.

Thus, a configuration is needed to use the platform- and compiler-specific features. For this purpose, we are using the GNU autoconf utility to automatically generate compiler-specific header file `ncbi-conf.h` and makefiles that would work for the given platform.

The user performs both configuration and installation by merely running platform-independent (*sh, bash*) shell script ***configure*** (we pre-generate it in-house from template `configure.ac` using autoconf).

During the configuration process, many compiler features are tested, and the results of this testing are recorded in the configuration header `ncbiconf.h` by the means of C preprocessor variables. For example, the preprocessor variable `NO_INCLASS_TMPL` indicates whether the compiler supports template class methods. Also contained in the `ncbiconf.h` file are preprocessor variables used to define sized integer and BigScalar types.

The ***configure*** script will create a build tree, a hierarchy of directories where object modules, libraries, and executables are to be built. It will also configure all `*.in` template files located in the NCBI C++ source tree (`src/`) and deploy the resultant configured files in the relevant places of the build tree. This way, all platform- and compiler-specific tools and flags will be "frozen" inside the configured makefiles in the build tree. The `ncbiconf.h` (described above, also configured for the given compiler) will be put to the `inc/` sub-directory of the resultant build tree.

User can create as many such *build trees* as needed. All these trees would refer to the same (origin) NCBI C++ source tree, whereas each build tree can contain its own (platform/compiler-specific) `ncbiconf.h` header and/or different set of compilation/linking flags and tools ("frozen" in the makefiles, particularly in Makefile.mk). This allows building libraries and executables using different compilers and/or flags, yet from the same source, and in a uniform way.

Structure of the Build Tree Produced by *configure*

Each configuration process results in a new *build tree*. The structure of the tree is (see also in chart):

`inc/` - contains the `ncbiconf.h` configuration header generated by the ***configure*** script.

`build/` - contains a hierarchy of directories that correspond to those in the `src/` (in NCBI C++ original sources). These directories will contain makefiles (`Makefile.*`) generated by the ***configure*** script from the makefile templates (`Makefile.*.in`) of the corresponding project located in the source tree. The resultant scripts and makefiles will keep references to the original NCBI C++ source directories. There is a "very special" file, Makefile.mk, that contains all configured tools, flags, and local paths. This file is usually included by other makefiles. Later, all build results (object modules, libraries, and executables, as well as any auxiliary files and directories created during the build) will go exclusively into the *build tree* and not to the original NCBI C++ source directories. This allows for several build trees to use the same source code while compiling and linking with different flags and/or compilers.

`lib/` - here go the libraries built by the `build/`-located projects.

`bin/` - here go the executables built by the `build/`-located projects.

`status/` - contains cache (`config.cache`), log file (`config.log`), and secondary configuration script (`config.status`) produced by ***configure*** during the configuration process.

Running the configure Script

The following topics are presented in this section:

Getting Synopsis of Available Configuration Options

Debug vs. Release Configuration

Building Shared Libraries (DLLs)

Hard-Coding Run-Time DLL Path into Executables and DLLs

Multi-Thread Safe Compilation and Linking with MT Libraries

Building in the 64-bit mode

Automatic Generation of Dependencies (for GNU make Only)

Naming the Build Tree

After-Configure User Callback Script

Tools and Flags

Localization for the System and Third-Party Packages

Prohibiting the Use of Some of the System and Third-party Packages

Optional Projects

Finer-grained Control of Projects: --with-projects

Miscellaneous: --without-exe, --without-execopy, --with-lib-rebuilds(=ask)

Compiler-specific configure Wrappers

Getting Synopsis of Available Configuration Options

To get the list of available configuration options (see Box 1), run `configure --help`. The NCBI-specific options are at the end of the printout. **NOTE:** Do not use the "standard" configure options listed in the *"Directory and file names:"* section of the help printout (such as `--prefix=`, `--bindir=`, etc.) because these are usually not used by the NCBI ***configure*** script.

Debug vs. Release Configuration

The following two ***configure*** flags control whether to target for the *Debug* or *Release* version. These options (default is *--with-debug*) control the appearance of preprocessor flags *-D_DEBUG* and *-DNDEBUG* and compiler/linker flags *-g* and *-O*, respectively:

--with-debug -- engage *-D_DEBUG* and *-g*, strip *-DNDEBUG* and *-O* (if not *--with-optimization*)

--without-debug -- strip *-D_DEBUG* and *-g*, engage *-DNDEBUG* and *-O* (if not *--without-optimization*)

--with-optimization -- unconditionally engage *-DNDEBUG* and *-O*

--without-optimization -- unconditionally strip *-DNDEBUG* and *-O*

default: *--with-debug --without-optimization*

Building Shared Libraries (DLLs)

On the capable platforms, you can build libraries as *shared (dynamic)*.

--*with-dll* --*with-static* -- build libraries as both *dynamic* and *static*; however, if the library project makefile specifies LIB_OR_DLL = lib, then build the library as *static* only, and if the library project makefile specifies LIB_OR_DLL = dll, then build the library as *dynamic* only. Note that the resulting static libraries consist of position-independent objects.

--*with-dll* -- build libraries as *dynamic*; however, if the library project makefile specifies LIB_OR_DLL = lib, then build the library as *static*

--*without-dll* -- always build *static* libraries, even if the library project makefile specifies LIB_OR_DLL = dll

default: build libraries as *static*; however, if the library project makefile specifies LIB_OR_DLL = dll, then build the library as *dynamic*

Hard-Coding Run-Time DLL Paths into Executables and DLLs

To be able to run executables linked against dynamic libraries (DLLs), you have to specify the location (runpath) of the DLLs. It can be done by hard-coding (using linker flags such as-*R.....*) the *runpath* into the executables.

--*with-runpath* - hard-code the path to the *lib/* dir of the Toolkit *build tree*.

--*with-runpath=/foo/bar* - hard-code the path to the user-defined */foo/bar* dir.

--*without-runpath* - do not hard-code any *runpath*.

default: if --*without-dll* flag is specified, then act as if --*without-runpath* was specified; otherwise, engage the --*with-runpath* scenario.

The preprocessor macro `NCBI_RUNPATH` will be set to the resulting runpath, if any.

NOTE: You also can use environment variable `$LD_LIBRARY_PATH` to specify the runpath, like this:

```
setenv LD_LIBRARY_PATH "/home/USERNAME/c++/WorkShop6-ReleaseDLL/lib"/home/USERNAME/c++/WorkShop6-
ReleaseDLL/bin/coretest
```

HINT: The --*with-runpath=....* option can be useful for the building of production DLLs and executables, which are meant to use production DLLs. The latter are usually installed not in the `lib/` dir of your development tree (*build tree*) but at some well-known dir of your production site. Thus, you can do the development in a "regular" manner (i.e., in a *build tree* configured using only --*with-runpath*); then, when you want to build a production version (which is to use, let's say, DLLs installed in `"/some_path/foo/"`), you must reconfigure your C++ build tree with just the same options as before, plus *"--with-runpath=/some_path/foo"*. Then rebuild the DLLs and executables and install

them into production. Then re-reconfigure your *build tree* back with its original flags (without the "*--with-runpath*=`/some_path/foo`") and continue with your development cycle, again using local in-tree DLLs.

Multi-Thread Safe Compilation and Linking with MT Libraries

--with-mt - compile all code in an MT-safe manner; link with the system thread library.

--without-mt - compile with no regard to MT safety.

default: *--without-mt*

Building in the 64-bit Mode

--with-64 - compile all code and build executables in the 64-bit mode.

default: depends on the platform; usually *--without-64* if both 32-bit and 64-bit build modes are available.

Automatic Generation of Dependencies (for GNU make only)

--with-autodep - add build rules to automatically generate dependencies for the compiled C/C++ sources.

--without-autodep - do not add these rules.

default: detect if the *make* command actually calls GNU *make*; if it does, then *--with-autodep*, else *--with-autodep*

Also, you can always switch between these two variants "manually", after the configuration is done, by setting the value of the variable Rules in Makefile.mk to either *rules* or `rules_with_autodep`.

NOTE: You **must** use GNU *make* if you configured with *--with-autodep*, because in this case the makefiles would use very specific GNU *make* features!

Naming the Build Tree

The configuration process will produce the new *build tree* in some default subdirectory of the root source directory, and the base name of this subdirectory will reflect the compiler name and a *Release/Debug* suffix, e.g., *GCC-Release/*. The default *build tree* name can be alternated by passing the following flags to the ***configure*** script:

--without-suffix - do not add *Release/Debug*, **MT**, and/or **DLL** suffix(es) to the *build tree* name. **Example:** *GCC/* instead of *GCC-ReleaseMT/*

--with-hostspec - add full host specs to the *build tree* name. **Example:** *GCC-Debug--i586-pc-linux-gnu/*

--with-build-root=/home/foo/bar - specify your own *build tree* path and name.

With *--with-build-root=*, you still can explicitly use *--with-suffix* and *--with-hostspec* to add suffix(s) to your *build tree* name in a manner described above. **Example:***--with-build-root=/home/foo/bar--with-mt --with-suffix* would deploy the new *build tree* in */home/foo/bar-DebugMT*.

There is also a special case with "*--with-build-root=.*" for those who prefer to put object files, libraries, and executables where the sources are located. But be advised that this will not allow you to configure other *build trees*.

After-Configure User Callback Script

You can specify your own script to call from the ***configure*** script after the configuration is complete:

--with-extra-action="<some_action>"

where <*some_action*> can be some script with parameters. The trick here is that in the <*some_action*> string, all occurrences of "**{}**" will be replaced by the build dir name.

Example:

```
configure --with-extra-action="echo foobar {}"
```

will execute (after the configuration is done):

```
echo foobar /home/user/c++/GCC-Debug
```

Tools and Flags

There is a predefined set of tools and flags used in the build process. The user can alternate these tools and flags by setting the environment variables shown in Table 1 for the ***configure*** script. In particular, if you intend to debug the Toolkit with Insure++, you should run ***configure*** with `CC` and `CXX` set to insure.

Later, these tools and flags will be engaged in the makefiles:

To compile C sources: `$(CC) -c $(CFLAGS) $(CPPFLAGS)....` .....

To compile C++ sources: `$(CXX) -c $(CXXFLAGS) $(CPPFLAGS)....` .....

To compose a library: `$(AR) libXXX.a xxx1.o xxx2.o xxx3.o .....$(RANLIB) libXXX.a`

To link an executable: `$(LINK) $(LDFLAGS) ..... $(LIBS)`

For more information on these and other variables, see the GNU autoconf documentation. The specified tools and flags will then be "frozen" inside the makefiles of *build tree* produced by this ***configure*** run.

Localization for the System and Third-Party Packages

There is some configuration info that usually cannot be guessed or detected automatically, and thus in most cases it must be specified "manually" for the given local host's working environment. The following localization environment variables can be set (see Table 2) in addition to the "generic" ones (CC, CXX, CPP, AR, RANLIB, STRIP, CFLAGS, CXXFLAGS, CPPFLAGS, LDFLAGS, LIBS):

On the basis of Table 2, ***configure*** will derive the variables shown in Table 3 to use in the generated makefiles.

Prohibiting the Use of Some of the System and Third-Party Packages

Some of the above system and third-party packages can be prohibited from use by using the following ***configure*** flags:

--without-sybase (Sybase)

--without-ftds (FreeTDS)

--without-fastcgi (FastCGI)

--without-fltk (FLTK)

--without-wxwin (wxWindows)

--without-ncbi-c (NCBI C Toolkit)

--without-sssdb (NCBI SSS DB)

--without-sssutils (NCBI SSS UTILS)

--without-sss (both *--without-sssdb* and *--without-sssutils*)

--without-geo (NCBI GEO)

--without-sp (NCBI SP)

--without-pubmed (NCBI PubMed)

--without-orbacus (ORBacus CORBA)

Optional Projects

You can control whether to build the following core packages using the following ***configure*** flags:

--*without-serial* -- do not build C++ ASN.1 serialization library and datatool; see in `internal/c++/{`src | include`}/serial` directories

--*without-ctools* -- do not build projects that use NCBI C Toolkit see in `internal/c++/{`src | include`}/ctools` directories

--*without-gui* -- do not build projects that use wxWindows GUI package see in `internal/c++/{`src | include`}/gui` directories

--*with-objects* -- generate and build libraries to serialize ASN.1 objects; see in `internal/c++/{`src | include`}/objects` directories

--*with-internal* -- build of internal projects is by default disabled on most platforms; see in `internal/c++/{`src | include`}/internal` directories

Finer-grained Control of Projects: --with-projects

If the above options aren't specific enough for you, you can also tell ***configure*** which projects you want to build by passing the flag --*with-projects=FILE*, where *FILE* contains a list of extended regular expressions indicating which directories to build in. With this option, the *make* target a`ll`_p will build all selected projects under the current directory. If there is a project you want to keep track of but not automatically build, you can follow its name with " update-only". To **exclude** projects that would otherwise match, list them explicitly with an initial hyphen. (Exclusions can also be regular expressions rather than simple project names.)

For instance, a file containing the lines

```
corelib$ util serial -serial/test test update-only
```

would request a non-recursive build in `corelib` and a recursive build in `util`, and a recursive build in `serial` that skipped `serial/test`. It would also request keeping the test project up-to-date (for the benefit of the programs in `util/test`).

NOTE: The flags listed above still apply; for instance, you still need --*with-internal* to enable internal projects. However, update_projects.sh can automatically take care of these for you; it will also take any lines starting with two hyphens as explicit options.

Miscellaneous: --without-exe, --without-execopy, --with-lib-rebuilds(=ask)

--*without-exe* -- do not build the executables enlisted in the `APP_PROJ`.

--*without-execopy* -- do not copy (yet build) the executables enlisted in the `APP_PROJ`.

--with-lib-rebuilds -- when building an application, attempt to rebuild all of the libraries it uses in case they are out of date.

--with-lib-rebuilds=ask -- as above, but prompt before any needed rebuilds. (Do not prompt for libraries that are up to date.)

Here's a more detailed explanation of *--with-lib-rebuilds*: There are three modes of operation:

In the default mode (*--without-lib-rebuilds*), starting a build from within a subtree (such as internal) will not attempt to build anything outside of that subtree.

In the unconditional mode (*--with-lib-rebuilds*), building an application will make the system rebuild any libraries it requires that are older than their sources. This can be useful if you have made a change that affects everything under objects but your project only needs a few of those libraries; in that case, you can save time by starting the build in your project's directory rather than at the top level.

The conditional mode (*--with-lib-rebuilds=ask*) is like the unconditional mode, except that when the system discovers that a needed library is out of date, it asks you about it. You can then choose between keeping your current version (because you prefer it or because nothing relevant has changed) and building an updated version.

Compiler-specific configure Wrappers

Most of the non-GCC compilers require special tools and additional mandatory flags to compile and link C++ code properly. That's why we have to have special scripts that perform the required non-standard compiler-specific pre-initialization for the tools and flags used before running ***configure***.

These wrapper scripts are located in the *compilers/* directory, and now we have such wrappers for the `SUN WorkShop` (`5.3`, `5.4`, and `5.5`), `MIPSpro 7.3`, `GCC` and `ICC` compilers:

- `WorkShop.sh` {32|64} [build_dir] [--configure-flags]
- `WorkShop55.sh` {32|64} [build_dir] [--configure-flags]
- `MIPSpro73.sh` {32|64} [build_dir] [--configure-flags]
- `ICC.sh` [build_dir] [--configure-flags]

Note that these scripts accept all regular ***configure*** flags and then pass them to the ***configure*** script.

Quick Reconfiguration

Sometimes, you change or add configurables (`*.in` files, such as *Makefile.in* meta-makefiles) in the *source tree*.

For the *build tree* to pick up these changes, go to the appropriate build directory and run the script ***reconfigure.sh***. It will automatically use just the same command-line arguments that you used for the original configuration of that *build tree*.

Run ***reconfigure.sh*** with argument:

update - if you did not add or remove any *configurables* in the *source tree* but only modified some of them.

reconf - if you changed, added, and/or removed any *configurables* in the *source tree*.

recheck - if you also suspect that your working environment (compiler features, accessibility of third-party packages, etc.) might have changed since your last (re)configuration of the *build tree* and, therefore, you do not want to use the cached check results obtained during the last (re)configuration.

without arguments - printout of script usage info.

Example:

```
cd /home/foobar/c++/GCC-Debug/build
./reconfigure.sh reconf
```

Naturally, *update* is the fastest of these methods, *reconf* is slower, and *recheck* (which is an exact equivalent of re-running the ***configure*** script with the same command-line arguments as were provided during the original configuration) is the slowest.

1. Environment variables that affect the build process

Name	Default	Synopsis
CC	gcc, cc	C compiler
CXX	c++, g++, gcc, CC, cxx, cc++	C++ compiler, also being used as a linker
CPP	$CC -E	C preprocessor
CXXCPP	$CXX -E	C++ preprocessor
AR	ar cru	Librarian
STRIP	strip	To discard symbolic info
CFLAGS	-g or/and/nor -O	C compiler flags
CXXFLAGS	-g or/and/nor -O	C++ compiler flags
CPPFLAGS	-D_DEBUG or/and/nor-DNDEBUG	C/C++ preprocessor flags
LDFLAGS	None	Linker flags
LIBS	None	Libraries to link to every executable
CONFIG_SHELL	/bin/sh	Command interpreter to use in the configuration scripts and makefiles (it must be compatible with sh)

2. User-defined localization variables.

Name	Default	Synopsis
THREAD_LIBS	-lpthread	System thread library
NETWORK_LIBS	-lsocket -lnsl	System network libraries
MATH_LIBS	-lm	System math library
KSTAT_LIBS	-lkstat	System kernel statistics library
RPCSVC_LIBS	-lrpcsvc	System RPC services library
CRYPT_LIBS	-lcrypt[_i]	System encrypting library
SYBASE_PATH	/netopt/Sybase/clients/current	Path to Sybase package (but see note below)
FTDS_PATH	/netopt/Sybase/clients-mssql/current	Path to FreeTDS package
FASTCGI_PATH	$NCBI/fcgi-current	Path to the in-house FastCGI client lib
FLTK_PATH	$NCBI/fltk	Path to the FLTK package
WXWIN_PATH	$NCBI/wxwin	Path to the wxWindows package
NCBI_C_PATH	$NCBI	Path to the NCBI C Toolkit
NCBI_SSS_PATH	$NCBI/sss/BUILD	Path to the NCBI SSS package
NCBI_GEO_PATH	$NCBI/geo	Path to the NCBI GEO package
SP_PATH	$NCBI/SP	Path to the SP package
NCBI_PM_PATH	$NCBI/pubmed[64]	Path to the NCBI PubMed package
ORBACUS_PATH	$NCBI/corba/OB-4.0.1	Path to the ORBacus CORBA package

Note: It is also possible to make configure look elsewhere for Sybase by means of --with-sybase-local[=DIR]. If you specify a directory, it will override SYBASE_PATH; otherwise, the default will change to /export/home/sybase/clients/current, but SYBASE_PATH will still take priority. Also, the option --with-sybase-new will change the default version of Sybase from 12.0 to 12.5 and adapt to its layout.
It is also possible to override WXWIN_PATH by --with-wxwin=DIR, FLTK_PATH by --> --with-fltk=DIR, and ORBACUS_PATH by --with-orbacus=DIR.

3. Derived localization variables for makefiles.

Name	Value	Used to...
THREAD_LIBS	$THREAD_LIBS	Link with system thread lib.
NETWORK_LIBS	$NETWORK_LIBS	Link with system network libs.
MATH_LIBS	$MATH_LIBS	Link with system math lib.
KSTAT_LIBS	$KSTAT_LIBS	Link with system kernel stat lib.
RPCSVC_LIBS	$RPCSVC_LIBS	Link with system RPC lib.
CRYPT_LIBS	$CRYPT_LIBS	Link with system encrypting lib.
SYBASE_INCLUDE	-I$SYBASE_PATH/include	#include Sybase headers
SYBASE_LIBS	-L$SYBASE_PATH/lib[64] -lblk[_r][64] -lct[_r][64] -lcs[_r][64] -ltcl[_r][64] -lcomn[_r][64] -lintl[_r][64]	Link with Sybase libs.
SYBASE_DLLS	-ltli[_r][64]	Sybase DLL-only libs
SYBASE_DBLIBS	-L$SYBASE_PATH/lib[64] -lsybdb[64]	Link with Sybase DB Lib API.
FTDS_INCLUDE	-I$FTDS_PATH/include	#include FreeTDS headers
FTDS_LIBS	-L$FTDS_PATH/lib -lsybdb -ltds	Link with the FreeTDS API.
FASTCGI_INCLUDE	-I$FASTCGI_PATH/include[64]	#include Fast-CGI headers
FASTCGI_LIBS	-L$FASTCGI_PATH/lib[64] -lfcgi or -L$FASTCGI_PATH/altlib[64] -lfcgi	Link with FastCGI lib.
FLTK_INCLUDE	-I$FLTK_PATH/include	#include FLTK headers
FLTK_LIBS	-L$FLTK_PATH/[GCC-]{Release\|Debug}[MT][64]/lib -lfltk ... -lXext -lX11 ... or -L$FLTK_PATH/lib	Link with FLTK libs.
WXWIN_INCLUDE	-I$WXWIN_PATH/include	#include wxWindows headers
WXWIN_LIBS	-L$WXWIN_PATH/[GCC-]{Release\|Debug}/lib -lwx_gtk[d] -lgtk -lgdk -lgmodule -lglib or -L$WXWIN_PATH/lib	Link with wxWindows libs.
NCBI_C_INCLUDE	-I$NCBI_C_PATH/include[64]	#include NCBI C Toolkit headers
NCBI_C_LIBPATH	-L$NCBI_C_PATH/lib[64] or -L$NCBI_C_PATH/altlib[64]	Path to NCBI C Toolkit libs.
NCBI_C_ncbi	-lncbi	NCBI C Toolkit CoreLib
NCBI_SSS_INCLUDE	-I$NCBI_SSS_PATH/include	#include NCBI SSS headers
NCBI_SSS_LIBPATH	-L$NCBI_SSS_PATH/lib/.... ....{Release|Debug}[GNU][64][mt]	Link with NCBI SSS libs.
NCBI_GEO_INCLUDE	-I$NCBI_GEO_PATH/include	#include NCBI GEO headers
NCBI_GEO_LIBPATH	-L$NCBI_GEO_PATH/lib/.... ...[GCC-|KCC-|ICC-]{Release|Debug}[64]	Link with NCBI GEO libs.
SP_INCLUDE	-I$SP_PATH/include	#include SP headers
SP_LIBS	-L$SP_PATH/{Release|Debug}[MT][64] -lsp	Link with the SP lib.
NCBI_PM_PATH	$NCBI_PM_PATH	Path to the PubMed package.
ORBACUS_INCLUDE	-I$ORBACUS_PATH/include -I$ORBACUS_PATH/{Release|Debug}[MT][64]/inc	#include ORBacus CORBA headers
ORBACUS_LIBPATH	-L$ORBACUS_PATH/{Release|Debug}[MT][64]/lib	Link with ORBacus CORBA libs.

MS Windows

- MS Visual C++.NET 7.1
- MS Visual C++.NET 8.0
- Cygwin/GCC

MS Visual C++.NET 7.1

The following topics are discussed in this section:

- Get the Toolkit sources
- Build external libraries (optional)
- Build the Toolkit
- The Build Results
- Create custom solution
- Start a new project that uses the Toolkit
- Modify existing project in the Toolkit
- Start a new project in the Toolkit
- Generic (UNIX and MS Visual Studio) build rules
- Site-specific Build Tree Configuration
- DLL Configuration
- Fine-Tuning MSVC Project Files
- Excluding project from the build
- Adding files to project
- Excluding files from project
- Adjusting build tools settings
- Specifying custom build rules

Get the Toolkit Sources

The NCBI C++ Toolkit sources can be either retrieved from CVS repository:

```
cvs checkout -d cxx internal/c++
```

or extracted from the publicly distributed archive.

Build External Libraries

Some of the NCBI C++ Toolkit projects make use of the NCBI C Toolkit and/or freely distributed 3rd-party libraries (such as BerkeleyDB, LibZ, FLTK, etc.).

At NCBI, these libraries are already installed, and their locations are hard coded in the C++ Toolkit configuration files.

Alternatively, the source code for the NCBI C Toolkit [ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/2005/Dec_31_2005/NCBI_C_Toolkit] and the 3rd-party packages [ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools++/2005/Dec_31_2005/ThirdParty] can be downloaded from the NCBI FTP site and built - ideally, in all available configurations.

Build the Toolkit

Project files and solutions for MS Visual Studio are generated by a special project **-CONFIGURE-,** using:

- cxx\src\....\Makefile.* (UNIX-style makefile templates)
- Visual Studio specific configuration files:
    - cxx\compilers\msvc710_prj\project_tree_builder.ini
    - cxx\compilers\msvc710_prj\dll\dll_info.ini
    - cxx\src\....\Makefile.*.msvc

The list of the projects to build is taken from cxx\scripts\projects*.lst.

There are 5 standard solutions:

```
cxx\compilers\msvc710_prj\static\build\ncbi_cpp.sln

cxx\compilers\msvc710_prj\static\build\gui\ncbi_gui.sln

cxx\compilers\msvc710_prj\dll\build\ncbi_cpp_dll.sln

cxx\compilers\msvc710_prj\dll\build\gui\ncbi_gui_dll.sln

cxx\compilers\msvc710_prj\dll\build\gbench\ncbi_gbench.sln
```

The first two solutions are to build libraries and applications using static runtime libraries, and the other three use DLL runtime libraries.

Initially, all solutions contain only two projects: **-CONFIGURE-** and **-CONFIGURE-DIALOG-**. They generate all other Visual Studio projects and regenerate the solution. **-CONFIGURE-DIALOG-** shows a popup dialog box that can be used to modify various configuration settings.

The **-CONFIGURE-** build rewrites the currently open solution. To ensure that all projects are reloaded correctly, close the solution after the **-CONFIGURE-** build, and then open it again.

Now, all Toolkit "real" projects can be built using **-BUILD-ALL-** project.

Optional customization of the build environment is provided using the 'user' solution template (`ncbi_user.sln`), the 'import project' (`import_project.wsf`) or the 'new project' (`new_project.wsf`) MS Windows scripts.

The Build Results

The built Toolkit applications and libraries will be put, respectively, to:

`cxx\compilers\msvc710_prj\{static,dll}\bin\<ConfigurationName>`

`cxx\compilers\msvc710_prj\{static,dll}\lib\<ConfigurationName>`

Create Custom Solution

To help jump-start a new, customized solution, there is (in addition to the 5 standard solutions) a template solution `cxx\compilers\msvc710_prj\user\build\ncbi_user.sln`. The list of folders that need to be included into this solution is defined in `cxx\scripts\projects\ncbi_user.lst`

N.B. Do not use this solution directly. Instead, make a copy of `cxx\compilers\msvc710_prj\user` subtree and `cxx\scripts\projects\ncbi_user.lst` file; then rename the copies of `ncbi_user.sln` and `ncbi_user.lst`, and edit these copies.

For example, assuming that these new copies are named user.sln and user.lst:

Specify the list of input folders in the `user.lst` file

Open solution `user.sln`

Edit `configure._` file in **-CONFIGURE-** project (change `ncbi_user.lst` to `user.lst`)

By default, the solution uses static runtime libraries. To use DLL ones, add '-dll' command line parameter right before the -logfile, like this: `"%PTB_PATH%\project_tree_builder.exe" -dll -logfile ....`

Now, build project **-CONFIGURE-** project.

Start a New Project That Uses the Toolkit

To use an already built C++ Toolkit (with all its build settings and configured paths), use script `cxx\scripts\new_project.wsf` to create a new project:

```
cscript \cxx\scripts\new_project.wsf <name> <type> [builddir] [flags]
```

Here, 'name' is the name of the project to create; 'type' is one of the predefined project types (run the script with no parameters to see the list of available types); 'builddir' is the location of the C++ Toolkit libraries. E.g. if the Toolkit is built in `U:\cxx` folder:

```
cscript U:\cxx\scripts\new_project.wsf test app U:\cxx\compilers\msvc710_prj
```

The script creates new local build tree, puts the project source files to `\src\name` folder, header files to `name\include\name`, Visual Studio project file to `name\compilers\msvc710_prj\static\build\name`, and the solution file to `name\compilers\msvc710_prj\static\build`.

To add new source files or libraries to the project, edit `name\src\name\Makefile.name.app` makefile template, then rebuild **-CONFIGURE-** project of the solution.

Modify Existing Project in the Toolkit

At NCBI, to work on just a few C++ Toolkit projects the 'import project.wsf' script can be used to avoid retrieving and building of the whole C++ Toolkit source tree. E.g. to work on '`corelib`' subtree, run:

```
cscript U:\cxx\scripts\import_project.wsf corelib
```

The script will create the build tree, copy (or extract from CVS) relevant files, and create Visual Studio project files and solution which reference pre-built Toolkit libraries installed elsewhere.

Start a New Project in the Toolkit

Follow the regular UNIX-style guidelines on the adding of new project into the Toolkit.

Then, build the **-CONFIGURE-** project and reload the solution.

To start a new project that will become part of the Toolkit, create makefile template first. For applications it must be named `Makefile.< project_name>.app`; for libraries - `Makefile.<project_name>.lib`. If it is a new folder in the source tree, you will also need `Makefile.in` file here - to describe to configuration system what should be built in this folder. Also, such new folder must be mentioned in the parent folder's `Makefile.in` SUB_PROJ section. Finally, make sure your new project folder is listed in the appropriate "`*.lst`" file in `cxx\scripts\projects` - it can be either a subdirectory of an already listed directory, or a new entry in the list.

Generic (UNIX and MS Visual Studio) build rules

- `cxx\src\....\Makefile.in`
- `cxx\src\....\Makefile.*.app`
- `cxx\src\....\Makefile.*.lib`

These `UNIX`-style makefile templates describe the overall hierarchical structure of the C++ Toolkit projects and also the individual projects' dependencies, source files and build parameters (flags). These files are used to generate build rules for both UNIX and MS Visual Studio.

Site-specific Build Tree Configuration

File project_tree_builder.ini(see Table 4) describes build and source tree configurations, contains information about the location of 3rd-party libraries and applications, and includes information used to resolve macro definitions found in the `UNIX`-style makefile templates.

Toolkit project makefiles can list (in the pseudo-macro REQUIRES) a set of requirements that must be met in order for the project to be built. For example, a project can be built only on `UNIX`, or only in multi-thread mode, or if a specific external library is available. Depending on which of the requirements are met, the Toolkit configurator may exclude some projects in some (or all) build configurations or define preprocessor and/or makefile macros.

Some of the Toolkit projects can be built differently depending on the availability of non-Toolkit components. For them, there is a list of macros - defined in '`Defines`' entry - that define conditional compilation. To establish a link between such macro and a specific component, configuration file also has sections with the names of the macro. For each build configuration project tree builder creates a header file (see '`DefinesPath`' entry) and defines these macros there depending on the availability of corresponding components.

Many of the requirements define dependency on the components that are 3rd-party packages, such as BerkeleyDB. For each one of these there is a special section (e.g. [BerkeleyDB]) in project_tree_builder.ini that describes the path(s) to the `include` and `library` directories of the package, as well as the preprocessor definitions to compile with and the libraries to link against. The Toolkit configurator checks if the package's directories and libraries do exist, and uses this information when generating appropriate MSVS projects.

There are a few indispensable external components that have their last-resort analog in the Toolkit. '`LibChoices`' entry identifies such pairs, and '`LibChoiceIncludes`' provides additional include paths to the builtin headers.

NOTE: There are some requirements which, when building for MS Visual Studio, are always or never met. These requirements are listed in '`ProvidedRequests`', '`StandardFeatures`', or '`NotProvidedRequests`' of '`Configure`' section.

DLL Configuration

The Toolkit `UNIX`-style makefile templates give a choice of building the library as DLL or static (or both). Oftentimes however it is convenient to assemble "bigger" DLL made of the sources of several static libraries.

The dll_info.ini (See Table 5) file specifies which libraries form such composite DLLs. The file begins with a list of DLLs to build - '`DLLs`' entry in '`DllBuild`' section. Then, there is a section with the name of each item from this list.

It defines:

- 'Hosting' - list of the static library projects whose sources will be included into this DLL
- 'Dependencies' - list of libraries that this DLL depends on
- 'DllDefine' - preprocessor macro that should be defined when compiling this DLL project. This macro is used to define Microsoft-specific storage-class modifiers, which export or import functions, data, and objects to and from DLL (see also include\corelib\mswin_export.h header file).

Fine-Tuning MSVC Project Files

While default MSVS project settings are defined in Makefile.mk.in.msvc file, each project can require additional MSVC-specific fine-tuning, such as compiler or linker options, additional source code, etc. These tune-ups can be specified in Makefile.<project_name>.[lib|app].msvc file located in the project source directory. All entries in such *.msvc file are optional.

Any section name can have one or several optional suffixes, so it can take the following forms:

- SectionName
- SectionName.[static|dll]
- SectionName.[debug|release]
- SectionName.[static|dll] [debug|release]
- SectionName.[debug|release].ConfigurationName
- SectionName.[static|dll] [debug|release].ConfigurationName

Here, 'static' or 'dll' means the type of runtime libraries that a particular build uses; 'debug' or 'release' - the type of the build configuration; 'ConfigurationName' - the name of the build configuration.

Settings in sections with more detailed names override ones in sections with less detailed names.

Excluding project from the build

To exclude project from the build, set 'ExcludeProject' entry in 'Common' section:

- [Common]
- ExcludeProject=TRUE

Adding files to project

This information should be entered in 'AddToProject' section. The section can have the following entries:

- [AddToProject]
- SourceFiles=
- ResourceFiles=
- IncludeDirs=

- LIB=
- HeadersInInclude=
- HeadersInSrc=

'SourceFiles' entry lists additional (usually MS Windows specific) source files (without extension) for the project. 'ResourceFiles' entry lists MS Windows resource files, 'IncludeDirs' - additional include directories, and 'LIB' - additional libraries for the project.

By default, all header files found in the project's include and source directories are added to the MSVS project. If that's not exactly what you need though, then the list of headers can be alternated using 'HeadersInInclude' and 'HeadersInSrc' entries. There, file names should be entered with their extension; exclamation mark means negation; wildcards are allowed. For example:

```
HeadersInInclude = *.h file1.hpp !file2.h
```

Means "add all files with h extension, add `file1.hpp`, and do not add `file2.h`"

NOTE: A single exclamation mark with no file name means "do not add any header files".

Excluding files from project

The information should be entered in 'ExcludeFromProject' section, which can have 'SourceFiles' and 'LIB' entries.

Adjusting build tools settings

Build tools are 'Compiler', 'Linker', 'Librarian', and 'ResourceCompiler' - that is, the tools used by MS Visual Studio build system. The names of available entries in any one of these sections can be found in `Makefile.mk.in.msvc` file, the meaning and possible values - in the description of "Microsoft Development Environment VC++ Project System Engine 7.0 Type Library".

Specifying custom build rules

To specify custom build rules for selected files in the project (usually non C++ files) use 'CustomBuild' section. It has a single entry - 'SourceFiles', which lists one or more files to apply the custom build rules to. Then, create a section with the name of the file, and define the following entries there: 'Commandline', 'Description', 'Outputs', and 'AdditionalDependencies' - that is, the same entries as in Custom Build Step of Microsoft Visual Studio project property pages. This data will then be inserted "as is" into the MSVS project file.

MS Visual C++.NET 8.0

To build the project on Visual C++.NET 8.0 follow the instructions for Visual C++.NET 7.1 Build the Toolkit and The Build Results replacing `msvc710_prj` with `msvc800_prj` respectively.

Note: The Visual C++.NET 8.0 seems to work significantly slower than Visual C++.NET 7.1.

Cygwin/GCC

To build the project on Cygwin/GCC just follow the generic Unix guidelines.

4. Project Tree Builder INI file (Local Site)

Section	Key	Comments
[ProjectTree]	MetaData	Makefile.mk.in - in this file the project tree builder will be looking for the UNIX project tree macro definitions.
	include	include "include" branch of project tree
	src	src "src" branch
	compilers	compilers "compilers" branch
	projects	scripts/projects "projects" branch
[msvc7]	Configurations	List of configurations to build
	compilers	Sub-branch of compilers branch for MSVC 7.10 projects
	MakefilesExt	Extension of MSVC-specific makefiles
	Projects	"build" sub-branch
	MetaMakefile	Master .msvc makefile - Makefile.mk.in.msvc
	DllInfo	INI file with DLLs information (hosting, dependencies, define for export prefixes)
[Configure]	NotProvidedRequests	List of requirements from UNIX makefiles that will not be provided. Projects with such requirements will not be created.
	DefinesPath	Path to .h file that will contain HAVE_XXXX definitions. The path is relative from the project tree root.
	Defines	List of HAVE_XXXX preprocessor definitions.
	LibChoices	List of pairs <libID>/<Component>. If the third-party library <Component> is present, then this library will be used instead of the internal library <libID>.
	ThirdPartyLibsToInstall	List of components, which DLLs will be automatically installed in the binary build directory.
	ThirdPartyLibsBinPathSuffix	Part of the naming convention for third-party DLLs installation makefile.
	ThirdPartyLibsBinSubDir	Part of the third-party DLLs installation target location.
[LibChoicesIncludes]	CMPRS_INCLUDE	Definition for the include directories for LibChoices.
[HAVE_XXXX]	Component	List of the components to check. An empty list means that the component is always available. A non-empty list means that the component(s) must be checked on presentation during configure.
[Debug],[DebugDLL],etc...	debug	TRUE means that the debug configuration will be created.
	runtimeLibraryOption	C++ Runtime library to use.
[NCBI_C_LIBS],[FLTK_LIBS_GL]	Component	List of libraries to use.
[<LIBRARY>]	INCLUDE	Include path to the library headers.
	DEFINES	Preprocessor definition for library usage.
	LIBPATH	Path to library.
	LIB	Library files.
	CONFS	List of supported configurations.
[Defines]	NCBI_C_INCLUDE	NCBI C Toolkit library include path.
	NCBI_C_ncbi	NCBI C Toolkit libraries to use in projects with this definition in the UNIX makefile.

Section	Key	Comments
[DefaultLibs]	INCLUDE	Default libraries will be added to each project. This section is to negotiate the differences in the default libraries on the UNIX and Win32 platforms. Same as for [<LIBRARY>].
	LIBPATH	Same as for [<LIBRARY>].
	LIB	Same as for [<LIBRARY>].
[Datatool]	datatool	ID of the datatool project. Some projects (with ASN or DTD sources) are dependent on the datatool.
	Location.App	Location of datatool executable for APP projects.
	Location.Lib	Location of datatool executable for LIB projects.
	CommandLine	Partial command line for datatool.

5. DLLs Information INI file (dll_info.ini)

Section	Key	Comments
[DllBuild]	DLLs	List of DLLs.
	Configurations	Configurations of DLL build.
	BuildDefine	Preprocessor definition for projects in DLL build.
[<dll_name>]	Hosting	List of libraries to host in this DLL.
	Dependencies	List of project IDs that this DLL depends upon.
	DllDefine	Preprocessor definition for the build libraries in DLL mode.

Mac OS X

Xcode

GCC

CodeWarrior

Xcode

Build the Toolkit

Open, build and run a project file in `compilers/xCode`.

This GUI tool generates a new NCBI C++ Toolkit Xcode project. It allows to:

- Choose which Toolkit libraries and applications to build.

- Automatically download and install all 3rd-party libraries.
- Specify third-party installation directories.

The Build Results

The built Toolkit applications and libraries will be put

to the output directory selected by the user during step A.

Apple Xcode versions 2.0 and above support build configurations. We use the default names Debug and Release, so the built applications will go to, for example:

- `<output_dir>/bin/Debug/Genome Workbench.app`
- `<output_dir>/bin/Release/Genome Workbench.app`

Apple Xcode versions before 2.0 do not support build configurations, so the build results will always go to:

```
<output_dir>/bin/Genome Workbench.app
```

Most libraries are built as Mach-O dynamically linked and shared (`.dylib`) and go to:

```
<output_dir>/lib
```

Genome Workbench plugins are built as Mach-O bundles (also with `.dylib` extension) and get placed inside Genome Workbench application bundle:

```
<output_dir>/Genome Workbench.app/Contents/MacOS/plugins
```

GCC

To build the project with GCC just follow the generic Unix guidelines.

CodeWarrior

For various reasons we have decided to drop support for CodeWarrior. The latest supported version of CodeWarrior can be found here